UiO : Faculty of Mathematics and Natural Sciences
University of Oslo
Department of Informatics
Networks and Distributed Systems (ND) group

ssthresh after slow start overshoot

Michael Welzl

ICCRG
IETF-120
26.7.2024

Disclaimer

- This is about a very basic aspect of TCP
  - and any protocol implementing its congestion control
- Partially "hidden" by modern TCP variants
  - Cubic exits with HyStart; BBR paces, which could be extra bad (!) for what I show, but it also exits without overshoot
- Still, good to understand, for new design
  - or when HyStart(++) or whatever new heuristic doesn't kick in...

Okay, so what is this about?

- **Why / when do double losses happen after slow start?**

- I made a model, and it shows: after slow start overshoot, `ssthresh = beta*cwnd` with `beta` >= 0.5 is often bad (`cwnd` or `FlightSize`, not the point !)
  - Yes, even 0.5, and pacing makes this worse!

- "Bad": **too large.** So, `beta` = 0.7 is even worse
  - ssthresh becoming too large means:
    after loss recovery, we will lose a packet again.
  - I suspect: this has contributed to the complexity of recovery
    (how to solve the recovery problem when the *target* is wrong ??)

Blasphemy !

- This quote justifies TCP Reno's choice of `beta` = 0.5:

  "*If you're starting, you know that half the current window size 'worked', i.e., that a window's worth of packets were exchanged with no drops (slow-start guarantees this).*"
  V. Jacobson, "Congestion avoidance and control", SIGCOMM '88.

- **... and it's wrong.**

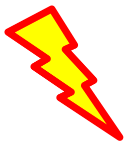
- In fact, when the the bottleneck is saturated, slow start guarantees that half the current window size is <u>too much</u>
  - cwnd was too large <u>and</u> it has grown before TCP learns about loss
  - TCP works, without HyStart etc., and without <u>always</u> having a loss after recovery, because bursts can cause earlier drops

An example

- BDP + queue = 30 packets

- IW=10: send 10 packets, then 20 (cwnd becomes 20, from 10 ACKs), then 40 (cwnd becomes 40, from 20 ACKs)

- Loss happens after 30 packets (losing #31)
  - The first 30 packets of this round go through
  - Their ACKs let cwnd grow to 40+30 = 70
  - Then, DupACKs arrive
  - 70 is the "current" window size now
  - 70/2 = 35

- 35 > 30 →  **another** packet loss.

A simple model to calculate ssthresh

- **Assumptions:**
  - One TCP connection, one bottleneck, FIFO queue
  - “Greedy” sender and no pacing *(but I'll show a pacing diagram)*
  - Connections are long enough to finish slow start
  - No link noise: packet loss is only caused by exceeding the queue
  - ACKs are neither delayed nor lost *(but I'll show a DelACK diagram)*

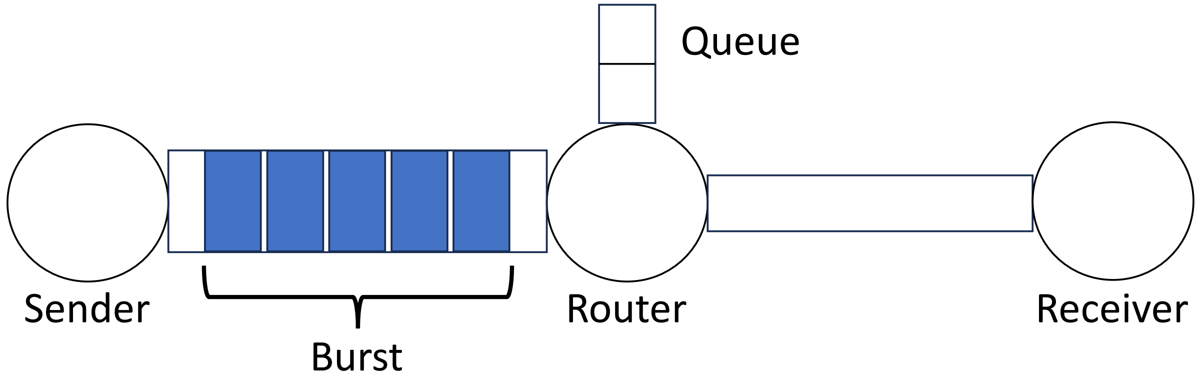
- Also: our unit is a packet; all packets have the same size

- So many constraints! Is this really useful?
  - Yes, this is how we understand TCP
  - When things go wrong, that's not great for more complex scenarios

Two types of packet loss

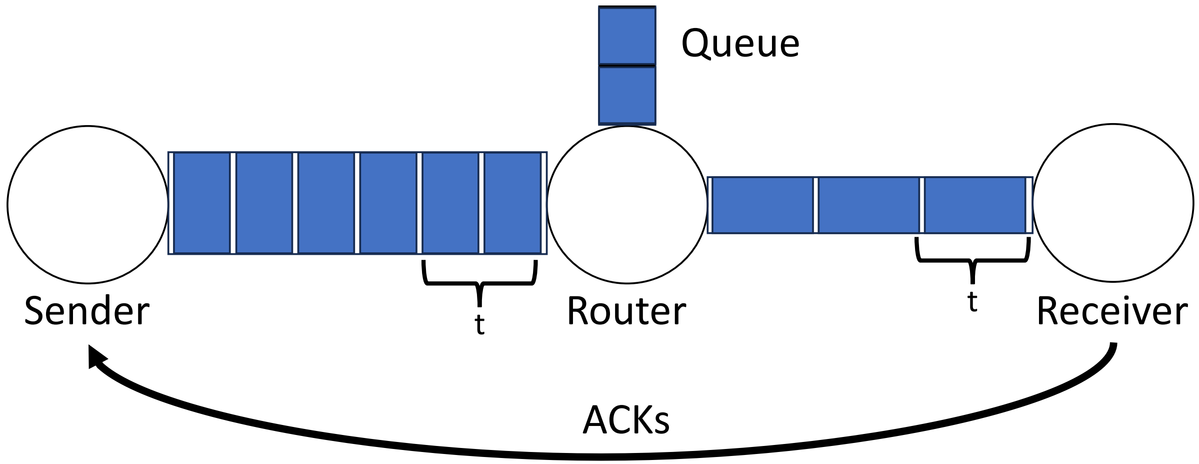
1. Burst loss

... a burst is too large. cwnd may be far below the BDP + queue limit!



2. Saturation loss

The bottleneck is truly "full".



Slow start ends when one of these two losses happens.

When does burst loss happen?

- When a burst is too large and arrives too fast
  - "Too fast": ratio departure / arrival rate **r**, **0 < r < 1**
  - Special case: capacities before the bottleneck ("ingress capacity") > 2 * capacity of bottleneck, and slow start: **r = 0.5**

- Burst **b**: when the last packet (packet **b**) arrives, **r(b-1)** packets have already been forwarded
  - E.g.: **r=0.5**, packet #5 arrives: 2 already forwarded.
  - and **q** (queue length in packets) packets can be queued
  - Constraint for a burst without overshoot:
    $b - \lfloor r(b-1) \rfloor - q <= 0$
    (rounding down because only complete packets are forwarded)
  - Making this equal to 0, and solving for **b**, yields...

The largest burst that doesn't cause loss

$$b = \left\lfloor \frac{q-r}{1-r} \right\rfloor$$

... and the smallest burst that causes loss: **b+1**.

Easy to test with IW

- Now let's relate this to TCP's slow start bursts
  - ACKs reflect the bottleneck's capacity
  - Doubling means, instead of **r**, we use **max(0.5, r)**
    - Why not simply 0.5? Just to cover cases where the ingress capacity is not at least twice the bottleneck's
  - Packets are sent as tuples of 2, with some space in between, but from the first to the last packet, the total burst's arrival rate > departure rate
    - So we can simplify and ignore this "micro-burstiness"

Slow start

- Initial window b_0
- In every round **i**, starting from 0, TCP sends $b_0\, 2^i$ packets
  - That's also the value that **cwnd** gets in this round!
- Let's assume no initial loss (i.e., $b+1 > b_0$), and call **k** the round in which loss happens. Then: $b + 1 <= b_0\, 2^k$
- We can turn this around and obtain **k**:

$$k = \left\lceil log_2 \left(\frac{q - r_i}{1 - r_i} + 1 \right) - log_2 \left(b_0 \right) \right\rceil, r_i = \begin{cases} r & \text{if } i = 0 \\ max(0.5, r) & \text{otherwise} \end{cases}$$

- No more need to round down for **b** because the round **k** is not influenced by fractions of packets
- Rounding up because, e.g., **k=1.3** means: the **k=1** burst was <u>not</u> large enough for loss → Loss occurs in round **k=2**

From a round to the cwnd

- We now know the number of the round: **k**
- In this round, **cwnd** will certainly become $\mathbf{b_0\,2^k}$
  - But, remember our BDP=30 example: this round's packets that pass through will let **cwnd** grow even further
  - These are **b** packets, so we get something like: $\mathbf{cwnd = b_0\,2^k + b}$
- It gets a bit more complex because of the ingress capacity constraint, and because for **k**, we assumed no initial loss, i.e. $\mathbf{b+1 > b_0}$
  - But we can use the equation for **k** to test for this condition: with $\mathbf{r_i=r}$, **is k zero?**

Getting to a final equation

- We now use **i** for the round in which the overshoot happens:

$$i = \begin{cases} \left\lceil log_2\left(\frac{q}{b_0}\right)\right\rceil + 1 & \text{if } k > 0 \text{ and } r \leq 0.5 \\ k & \text{if } k > 0 \text{ and } r > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

- With **i**, we define **r_i**:

$$r_i = \begin{cases} r & \text{if } i = 0 \\ \max(0.5, r) & \text{otherwise} \end{cases}$$

... and now, at last, we can write:

$$cwnd\_max_{burst} = b_0 2^i + \left\lfloor \frac{q - r_i}{1 - r_i} \right\rfloor$$

Simplifying

- The previous equation depends on **i**, which depends on **k**. We can make this easier:

$$cwnd\_max_{burst} = \begin{cases} b_0 + \left\lfloor \frac{q-r}{1-r} \right\rfloor & \text{if } i = 0 \\ b_0 2^{\left\lceil log_2\left(\frac{q}{b_0}\right)\right\rceil + 1} + 2q - 1 & \text{if } i > 0 \text{ and } r \leq 0.5 \\ b_0 2^i + \left\lfloor \frac{q-r_i}{1-r_i} \right\rfloor & \text{otherwise} \end{cases}$$

A very special case:
No loss in the first round <u>and</u> ingress capacity is not at least twice the bottleneck's

- Now, only the last case depends on i
- Else: loss in round 0: depends on $\mathbf{b_0}$, **q** and **r**
- Later loss: depends only on $\mathbf{b_0}$ and **q**

Including saturation loss is easy

$$cwnd\_max_{saturation} = 2\left(BDP + q\right)$$

$$cwnd\_max = \min\left(cwnd\_max_{saturation}, cwnd\_max_{burst}\right)$$

$$ssthresh = \beta \times cwnd\_max$$

Simplified, if loss does not happen in the first round (initial window) and the ingress capacity is at least twice the bottleneck's:

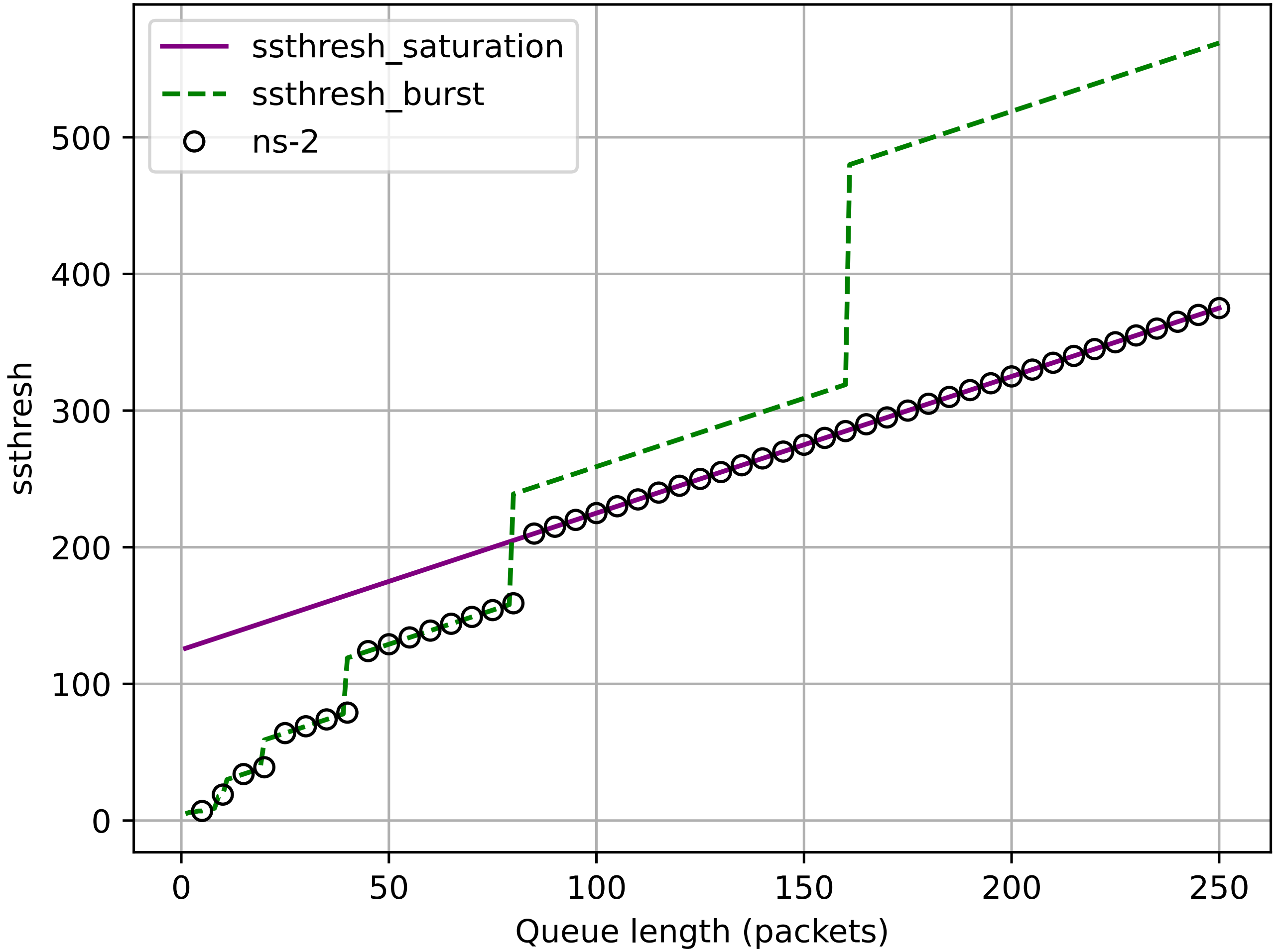
$$ssthresh_{not\_first\_round, r \leq 0.5} = \beta \times \min\left(2\left(BDP + q\right), b_0 2^{\left\lceil log_2\left(\frac{q}{b_0}\right)\right\rceil + 1} + 2q - 1\right)$$

So much theory!

Let's get real.

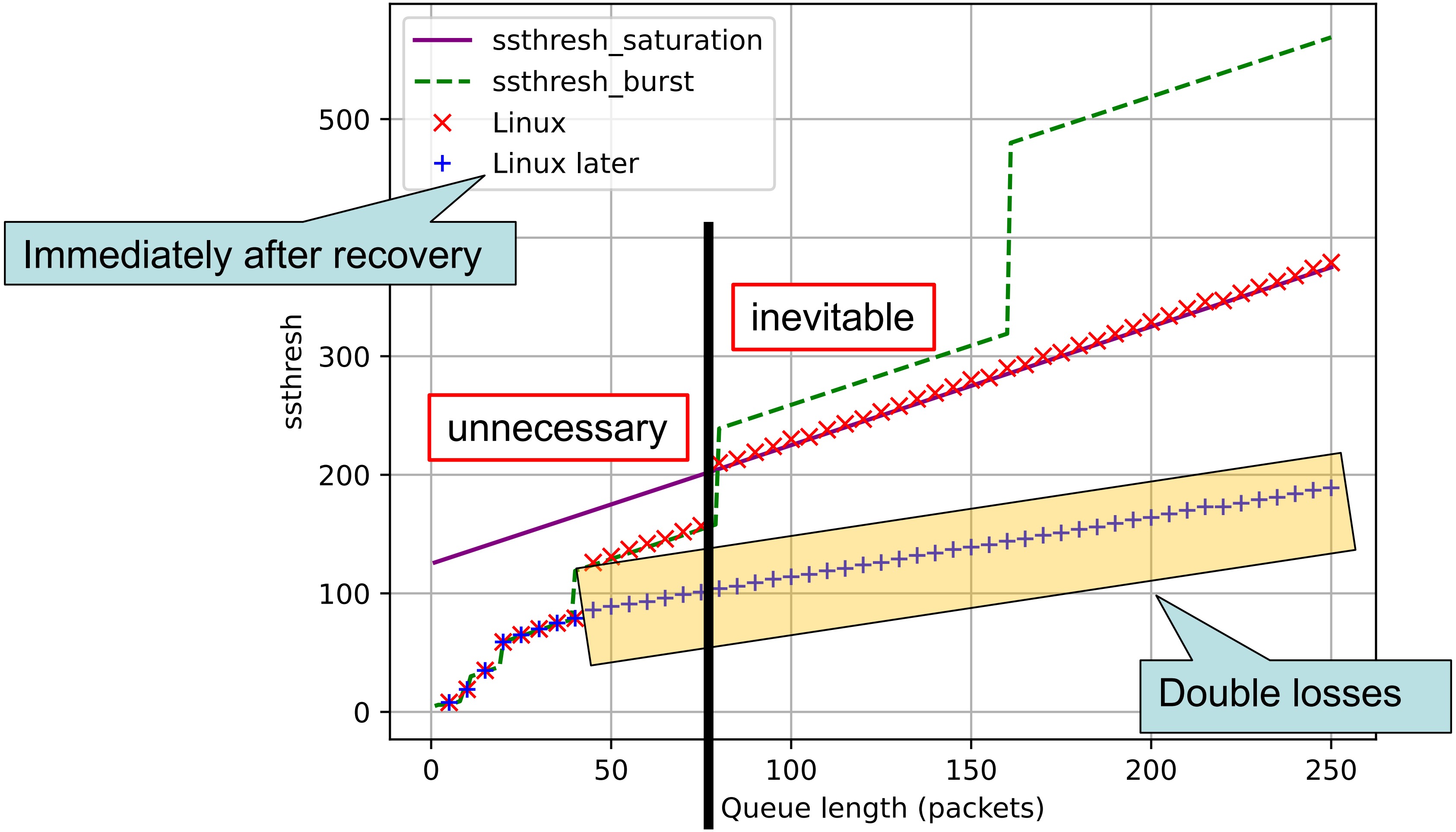
Side note: for all following diagrams, the queue length was varied from 0 to 2 BDPs

ns-2 validation



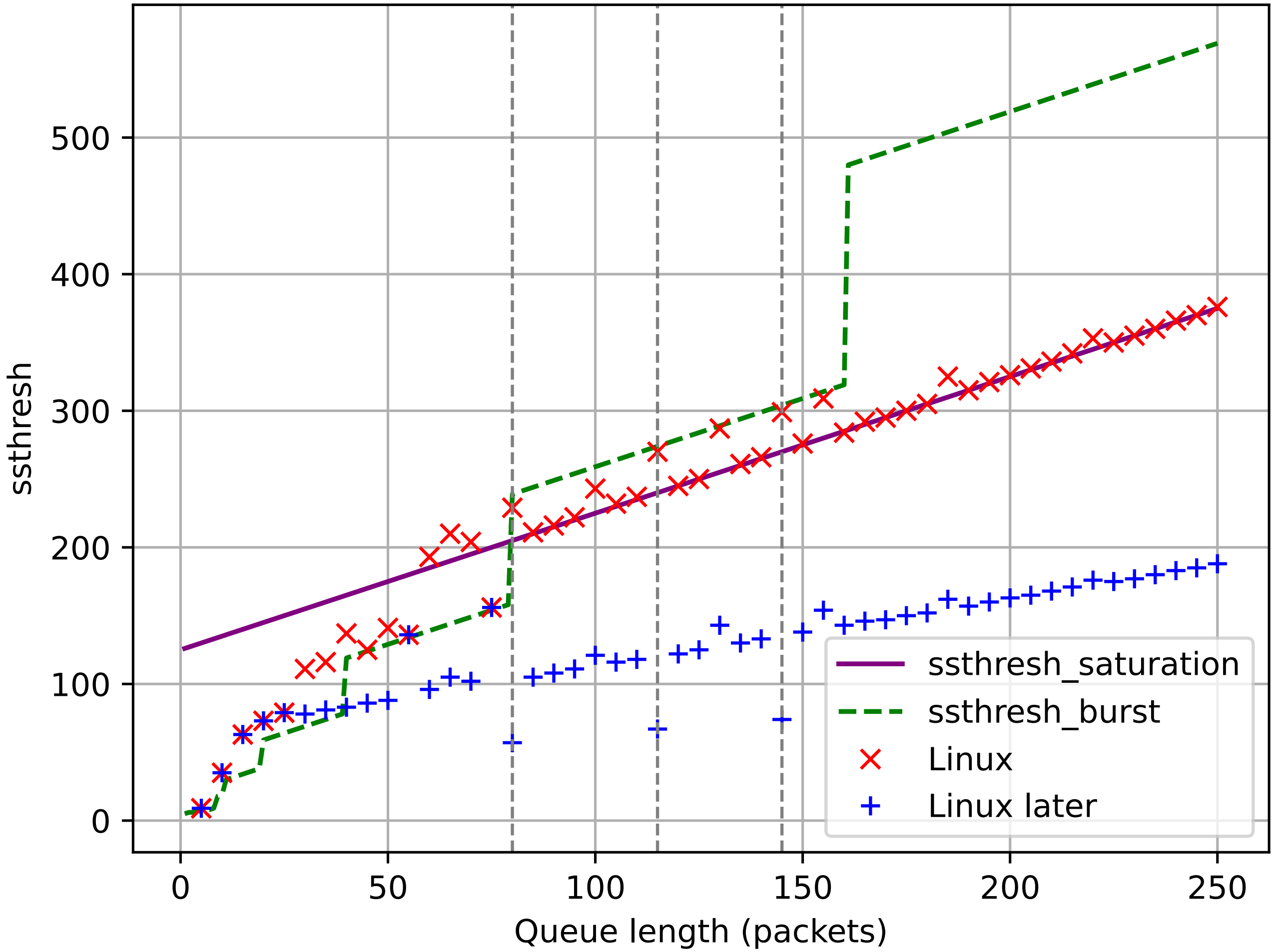
Ingress capacity 10 Gbit/s, bottleneck 50 Mbit/s, 30 ms RTT, IW 10
BDP = 125 packets

Linux validation



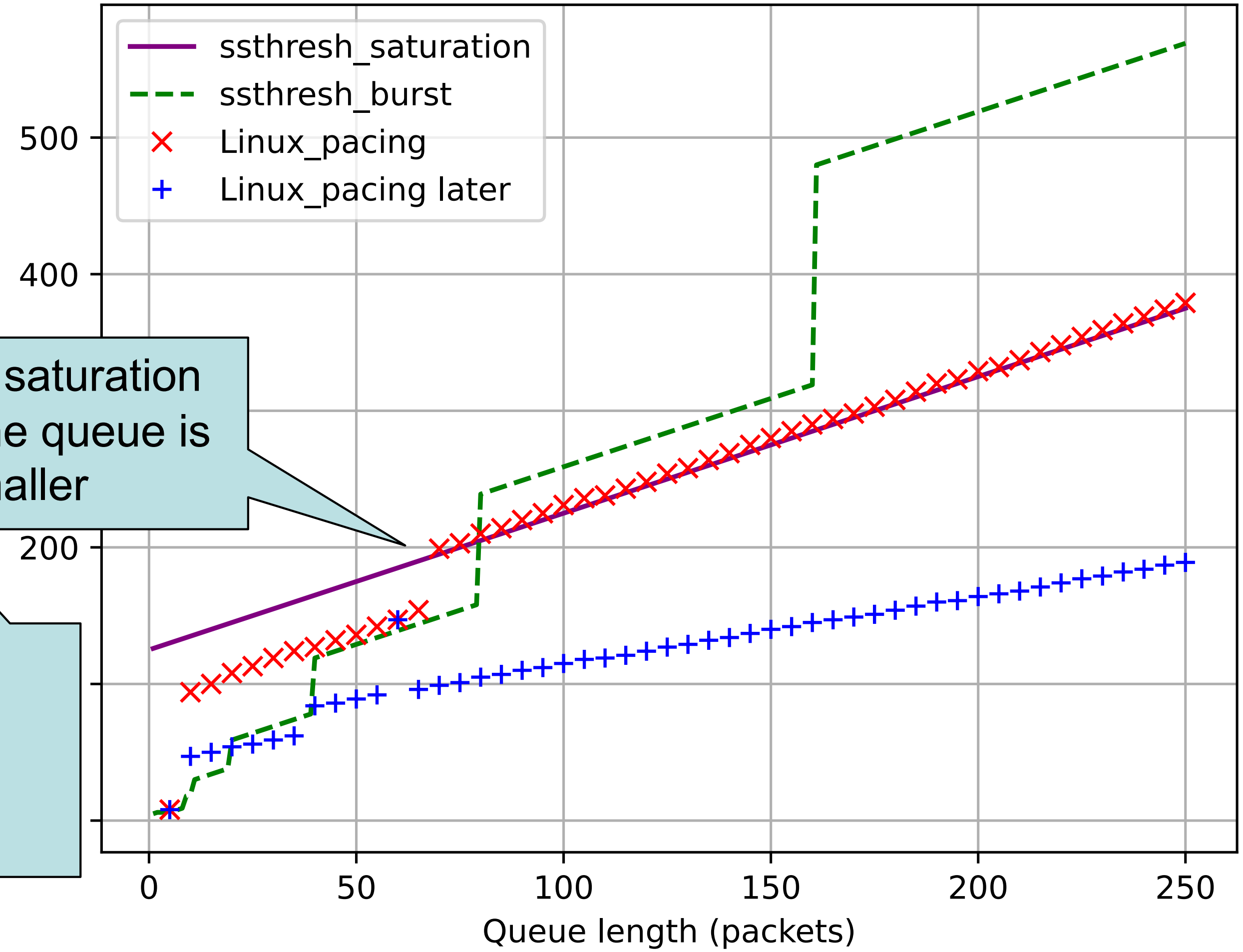
Ingress capacity 10 Gbit/s, bottleneck 50 Mbit/s, 30 ms RTT, IW 10
BDP = 125 packets

Linux validation with delayed ACKs



Ingress capacity 10 Gbit/s, bottleneck 50 Mbit/s, 30 ms RTT, IW 10
BDP = 125 packets

Linux validation with pacing

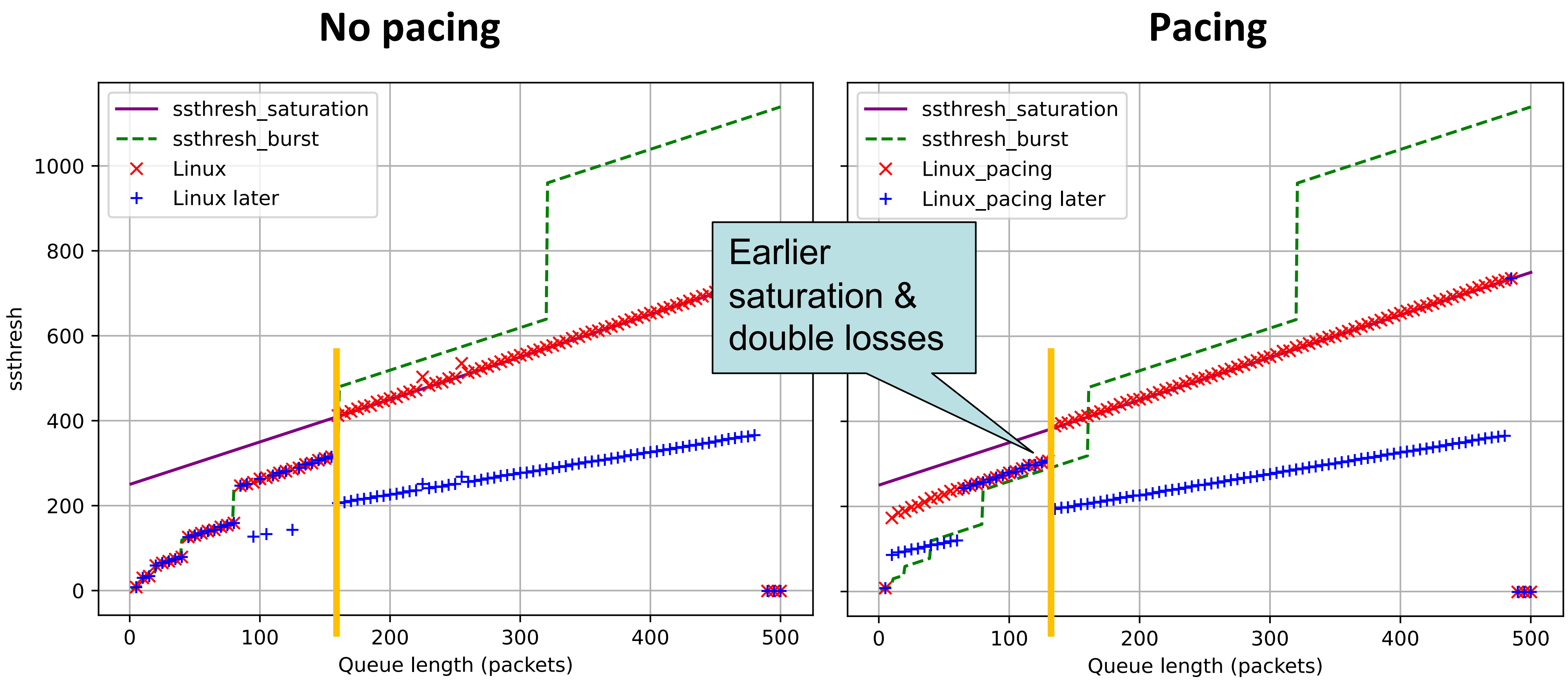


Gets to saturation when the queue is a bit smaller

... which guarantees a double loss with beta >= 0.5

Ingress capacity 10 Gbit/s, bottleneck 50 Mbit/s, 30 ms RTT, IW 10
BDP = 125 packets

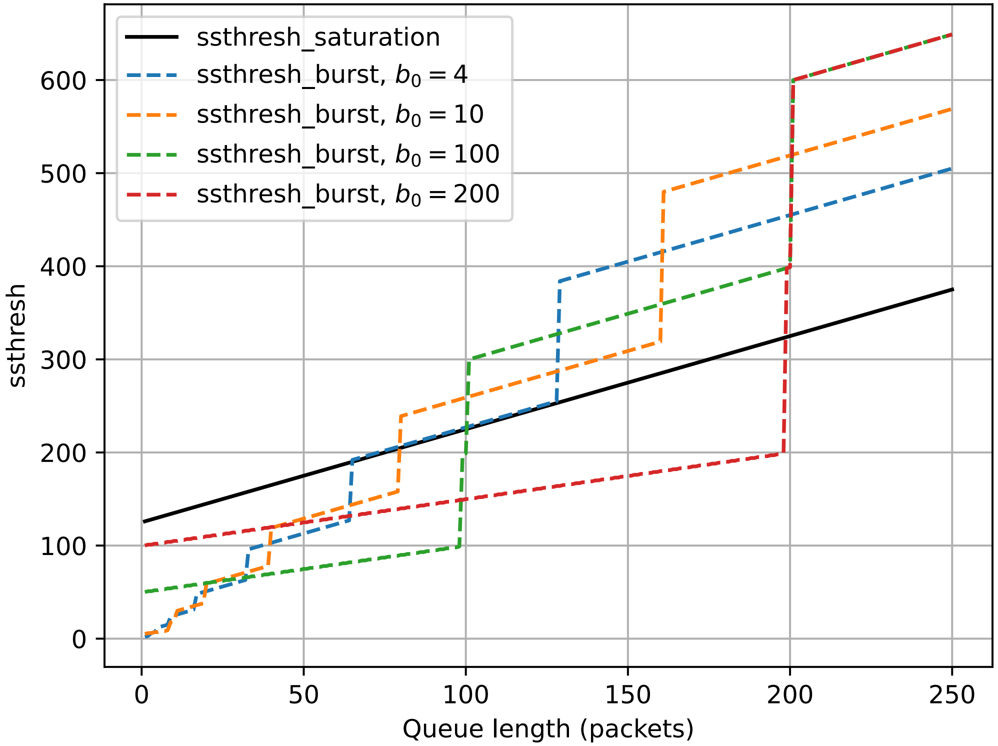
Linux validation with twice the BDP



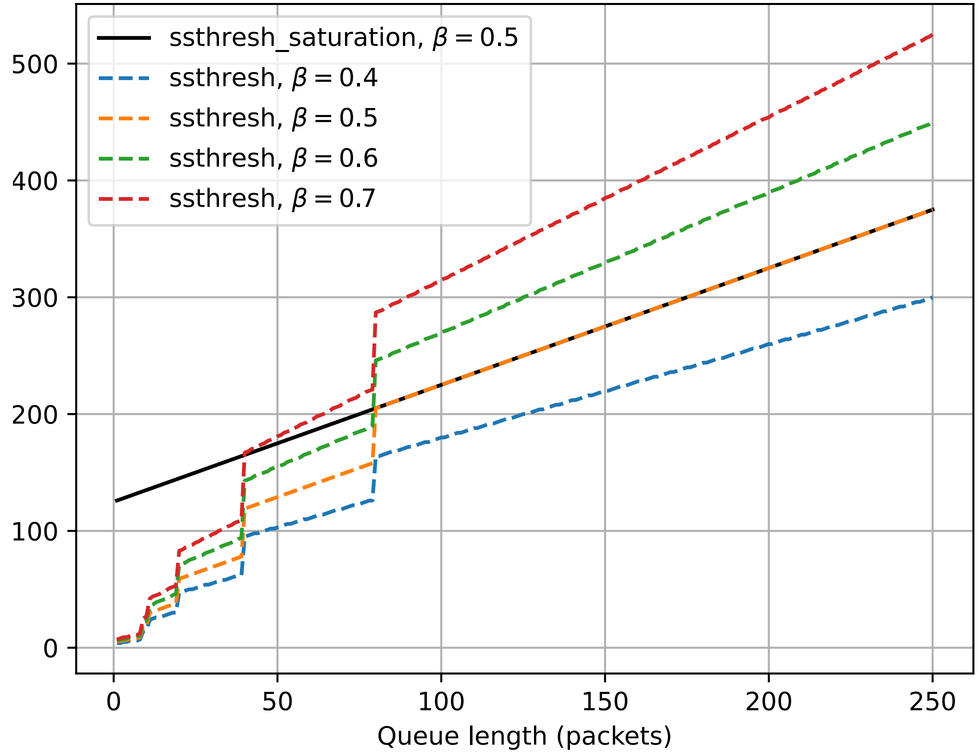
Ingress capacity 10 Gbit/s, bottleneck 50 Mbit/s, 60 ms RTT, IW 10
BDP = 250 packets

Model dependence on parameters

Initial window



beta



Ingress capacity 10 Gbit/s, bottleneck 50 Mbit/s, 30 ms RTT, IW 10
BDP = 125 packets

What is this all good for? Some ideas...

- The better pacing becomes, the "earlier" (lower queue) it will reach the saturation line
  - and the more likely it is that beta >= 0.5 is bad after slow start
- Heuristics could be designed...
  - E.g., to check which line we're on
  - E.g., to decide about beta or initial window from history
- Could adapt this for ECN
- Please don't do these things without me ☺

Beyond classic slow start

- The presented logic applies to overshoot in general
  - Relevant whenever it's more than just Reno's 1 packet

- Example: Conservative Slow Start (CSS) of HyStart ++
  - Like slow start, but with a smaller exponential base
  - This probably affects **r**, and "**2**" in this equation: $\mathbf{cwnd = b_0\ 2^k}$
  - So, probably no longer **0.5** in $\mathbf{r_i}$ here: $$r_i = \begin{cases} r & \text{if } i = 0 \\ \max(0.5, r) & \text{otherwise} \end{cases}$$
  - and in the final equation, $\mathbf{log_2}$ probably becomes $\mathbf{log_{something\_else}}$
  - I would have liked to look further, but I couldn't find an open source implementation, so couldn't easily validate it
  - I don't like mathematical modeling without validation

Thank you!

Questions?